Annals of Mathematics and Artificial Intelligence (2005) 44: 419–436 © Springer 2005

Improving exact algorithms for MAX-2-SAT *

Haiou Shen and Hantao Zhang

Computer Science Department, The University of Iowa, Iowa City, IA 52242, USA
E-mail: {hshen;hzhang}@cs.uiowa.edu

We study three new techniques that will speed up the branch-and-bound algorithm for the MAX-2-SAT problem: The first technique is a group of new lower bound functions for the algorithm and we show that these functions are admissible and consistently better than other known lower bound functions. The other two techniques are based on the strongly connected components of the implication graph of a 2CNF formula: One uses the graph to simplify the formula and the other uses the graph to design a new variable ordering. The experiments show that the simplification can reduce the size of the input substantially no matter what is the clause-to-variable ratio and that the new variable ordering performs much better when the clause-to-variable ratio is less than 2. A direct outcome of this research is a high-performance implementation of an exact algorithm for MAX-2-SAT which outperforms any implementation we know about in the same category. We also show that our implementation is a feasible and effective tool to solve large instances of the Max-Cut problem in graph theory.

Keywords: propositional satisfiability, maximum satisfiability, exact algorithms

1. Introduction

In recent years, there has been considerable interest in the maximum satisfiability problem (MAX-SAT) of propositional logic. Given a set of propositional clauses, MAX-SAT asks to find a truth assignment that satisfies the maximum number of clauses. The decision version of MAX-SAT is NP-complete, even if the clauses have at most two literals (so called the MAX-2-SAT problem). Because the MAX-SAT problem is fundamental to many practical problems in computer science [14] and computer engineering [23], efficient methods that can solve a large set of instances of MAX-SAT are eagerly sought. One important application of MAX-2-SAT is that NP-complete graph problems such as *maximum cut* and *independent set*, can be reduced to special instances of MAX-2-SAT [7,17]. Many of the proposed methods for MAX-SAT are based on approximation algorithms [8]; some of them are based on branch-and-bound methods [4,5,11–16,18]; and some of them are based on transforming MAX-SAT into SAT [23].

To the best of our knowledge, there are six implementations of exact algorithms for MAX-SAT that are variants of the well known Davis–Putnam–Logemann–Loveland

* Preliminary results of this paper appeared in [20,21]. This research was supported in part by NSF under grant CCR-0098093.

(DPLL) procedure [9]. One is due to Wallace and Freuder (implemented in Lisp) [22]; one is due to Gramm [11]; one is due to Borchers and Furman [5] (implemented in C and publicly available); the next two are made available in 2003 by Alsinet et al. [1] (a substantial improvement over Borchers and Furman's implementation) and Zhang et al. [24], respectively; the last one is provided by Zhao and Zhang in 2004 [25].

In this paper we will discuss three novel techniques intended to improve the performance of the branch-and-bound algorithm proposed in [24]. It is well known that the tighter the bound the smaller the search tree in a typical branch-and-bound algorithm. It is not a surprise to see that Alsinet et al.'s implementation is better than Borchers and Furman's implementation because a better lower bound function is used in [1]. After studying these lower bound functions, we introduce a group of new lower bound functions which can reduce the search tree substantially and improve the performance of the branch-and-bound algorithm for MAX-2-SAT. We show that these new lower bound functions are admissible and provide tighter bounds than any known lower bound functions.

The other techniques are based on the use of the strongly connected components (SCC) in the implication graph of a 2CNF instance. It is well known that the satisfiability of 2CNF formula can be decided in linear time by computing SCC [3]. For MAX-2-SAT, we found that computing SCC can help us to (a) simplify the input greatly, and (b) design a new variable ordering for the algorithm in [24] to solve MAX-2-SAT.

In order to evaluate the new techniques, we present experimental results on thousands of MAX-2-SAT instances. We show that the improved algorithm is consistently and substantially better than all the known algorithms [1,5,24,25]. We also show the performance of the algorithm on a large set of random MAX-CUT problems.

2. Preliminaries

Let F be a formula in CNF with n variables $V = \{x_1, \ldots, x_n\}$ and m clauses. An *assignment* is a mapping from V to $\{0, 1\}$ and may be represented by a vector $\vec{X} \in \{0, 1\}^n$, where 0 means false and 1 means true. Let $S(\vec{X}, F)$ be the number of clauses satisfied by $\vec{X}$ and $K(F)$ be the minimal number of false clauses under any assignment. That is,

$$K(F) = m - \max\bigl\{S\bigl(\vec{X}, F\bigr) \mid \vec{X} \in \{0, 1\}^n\bigr\}.$$

If we use the right-hand side of the above equation for computing $K(F)$, we obtain a naive enumeration algorithm for MAX-SAT. $K(F)$ can also be computed by a simple recursive function as follows:

$$K(F) = \min\bigl(K(F_x), K(F_{\bar{x}})\bigr)$$

where F_x is the formula obtained from F by replacing x by 1 and $\bar{x}$ by 0. The algorithm based on the above equation is also called Davis–Putanm–Logemann–Loveland procedure [9].

For every literal x, we use variable(x) to denote the variable appearing in x. That is, variable(x) $= x$ for positive literal x and variable($\bar{x}$) $= x$ for negative literal $\bar{x}$. If y is a literal, we use $\bar{y}$ to denote x if $y = \bar{x}$. A literal x in a set F of clauses is said to be *pure* in F if its complement does not appear in F.

A partial (complete) assignment can be represented by a set of literals in which each variable appears at most (exactly) once and each literal is meant to be true in the assignment. If variable(x) does not appear in a partial assignment A, then we say literals x and $\bar{x}$ are *unassigned* in A. For an unassigned literal x, $\mu(x)$ is the number of unit clauses for literal x in the current clause set, including those generated during the search. Initially, $\mu(x)$ is the number of unit clauses for literal x in the input.

The algorithm that we will use throughout the paper is presented in figure 1. This algorithm uses a fixed ordering, i.e., from x_1 to x_n, to assign truth value to variables (assuming $x_i < x_j$ if $i < j$). In the algorithm, $B(x) = \{y \mid (x \vee y) \in F, \text{variable}(x) < \text{variable}(y)\}$ for each literal x. By abuse of notations, we assume x_i is represented by i and $\bar{x}_i$ is represented by $\bar{i}$, where $1 \leqslant i \leqslant n$. Hence, we have $\mu(i) = \mu(x_i)$ and $B(i) = B(x_i)$.

The following result is provided in [24].

Theorem 2.1. Suppose F is a set of binary clauses on n variables. Then max_2_sat(F, n, g_0) returns true if and only if there exists an assignment under which at most g_0 clauses in F are false. The time complexity of max_2_sat(F, n, g_0) is $O(n2^n)$ and the space complexity is $L/2 + O(n)$, where L is the size of the input.

3. Lower bounds

It is well known that the tighter the bound the smaller the search tree in a typical branch-and-bound algorithm. In line 2 of dec_max_2_sat in figure 1, popular lower bound functions can be used to improve its performance. The following two lower bound functions are used in [1,2,24]:

- LB1 = the number of conflicting (or empty) clauses by the current partial assignment;
- LB2 = LB1 + $\sum_{j=i}^{n} \min(\mu(\bar{j}), \mu(j))$,

where $\mu(x)$ is the number of unit clauses for literal x under the current partial assignment. Using LB2 instead LB1 contributes greatly to the improved performance of Alsinet, Manyà and Planes' implementation over Borchers and Furman's. It is easy to see that LB1 $\leqslant$ LB2 $\leqslant$ $K(F)$. In general, a lower bound LB is said to be *admissible* if LB $\leqslant$ $K(F)$. In other words, both LB1 and LB2 are admissible.

Lemma 3.1. If there is a clause $x \vee y$ in F such that $\mu(x) < \mu(\bar{x})$ and $\mu(y) < \mu(\bar{y})$, then LB2 + 1 $\leqslant$ $K(F)$.

Proof. Note that LB2 = LB1 + $\sum_{j=i}^{n} \min(\mu(\bar{j}), \mu(j))$. If the assignments of x_j, where $i \leqslant j \leqslant n$, do not create any new unit clauses, then $K(F)$ = LB2. For a clause $x \vee y$,

```
function max_2_sat ( F: clause set, n, g_0: integer) return Boolean
  // initiation
  for i := 1 to n do
    compute B(i) and B(ī) from F;
    μ(i) := μ(ī) := 0; // assuming no unit clauses in F
  end for
  return dec_max_2_sat(1, g_0);
end function

function dec_max_2_sat( i, g: integer ) return Boolean
1  if (i > n) return true; // end of the search tree
2  if (lower_bound(i) > g) return false
3  // decide if we want to set variable i to true
4  if (μ(ī) ⩽ g) ∧ (μ(ī) < μ(i) + |B(i)|) then
5     record_unit_clauses(ī);
6     if (dec_max_2_sat(i + 1, g − μ(ī))) return true;
7     undo_record_unit_clauses(ī);
8  end if
9  // decide if we want to set variable i to false
10 if (μ(i) ⩽ g) ∧ (μ(i) ⩽ μ(ī) + |B(ī)|) then
11    record_unit_clauses(i);
12    if (dec_max_2_sat(i + 1, g − μ(i))) return true;
13    undo_record_unit_clauses(i);
14 end if
15 return false;
end function

procedure record_unit_clauses ( x: literal )
  for y ∈ B(x) do μ(y) := μ(y) + 1 end for;
end procedure

procedure undo_record_unit_clauses ( x: literal )
  for y ∈ B(x) do μ(y) := μ(y) − 1 end for;
end procedure
```

Figure 1. A decision algorithm for MAX-2-SAT.

if we assign 0 to x, $\mu(y)$ will be increased by 1. So $\min(\mu(y), \mu(\bar{y}))$ will be increased by 1. If we assign 1 to x, then we create $\mu(\bar{x})$ instead of $\mu(x)$ empty clauses, where $\mu(x) < \mu(\bar{x})$. So the number of empty clauses is greater than LB2. The same reason applies to y, too. □

```
function lower_bound3 ( i: integer) return integer
1  LB3 = LB2;
2  for every clause (x ∨ y) ∈ S(i) do
3      if (c(x) > 0) ∧ (c(y) > 0) then
4        LB3 = LB3 + 1; c(x) = c(x) − 1; c(y) = c(y) − 1;
5      end if
6  end for
12 return LB3
end function

function lower_bound4 ( i: integer) return integer
1  LB4 = LB1;
2  while (i ⩽ n) do
3      LB4 = LB4 + min(μ(ī), μ(i));
4      t = |μ(ī) − μ(i)|;
5      if μ(ī) > μ(i) then Y = B(i) else Y = B(ī)
6      for y ∈ Y if (t > 0) do
7        t = t − 1;
8        μ(y) = μ(y) + 1;
9      end for
10     i = i + 1
11 end while
12 return LB4
end function
```

Figure 2. The procedures for computing LB3 and LB4.

3.1. Lower bound LB3

The above lemma allows us to design an enhanced lower bound function as follows. For any literal x, let $c(x) = \mu(\bar{x}) - \mu(x)$ and $S(i)$ be the current set of clauses of which both literals are unassigned under the current assignment for variables x_1 to x_{i-1}. Then the new lower bound can be computed by the procedure lower_bound3 in figure 2.

It is easy to prove the following result.

Lemma 3.2. The worst-case time complexity of lower_bound3(i) is $O((n - i) + \sum_{j=i}^{n}(|B(j)| + |B(\bar{j})|))$.

Theorem 3.3. $LB2 \leqslant LB3 \leqslant K(F)$.

Proof. It is trivial to see $LB2 \leqslant LB3$. To prove that LB3 is admissible, we may apply lemma 3.1 for each execution of line 4. This is because $c(x) > 0$ and $c(y) > 0$ imply $\mu(\bar{x}) > \mu(x)$ and $\mu(\bar{y}) > \mu(y)$ and the conditions of lemma 3.1 hold for each execution. □

Note that to use lower_bound3 in dec_max_2_sat of figure 1, we need to replace the function lower_bound(i) in line 2 by lower_bound3(i) − LB1. This has to be done for other lower bound functions in this section as well, including LB2. We will see that LB3 improves the performance of the algorithm more than 10 times faster on average than LB2 on random MAX-2-SAT problems of 50 and 100 variables (see figures 4 and 5).

3.2. *Lower bound LB4*

While we are satisfied with the performance of LB3, we want to design another lower bound, lower_bound4 in figure 2, to fully explore the property revealed by lemma 3.1.

It is easy to prove the following result.

Lemma 3.4. The worst-case time complexity of lower_bound4(i) is $O((n - i) + \sum_{j=i}^{n} \max(|B(j)|, |B(\bar{j})|))$.

Before we prove formally that LB4 is admissible, let us look at a simple example.

Example 3.5. Suppose a 2CNF formula F is represented by μ and S, where $\mu(a) = \mu(b) = \mu(c) = \mu(\bar{c}) = 0$, $\mu(\bar{a}) = \mu(\bar{b}) = 1$ and $S = \{a \vee c, b \vee \bar{c}\}$. It is easy to see $K(F) = 1$. LB3 $= 0$ because there exists no clause $x \vee y$ in F such that $\mu(\bar{x}) > \mu(x)$ and $\mu(\bar{y}) > \mu(y)$.

Suppose $a < b < c$, then LB4 $= 1$ because $a \vee c$ will increase $\mu(c)$ by 1 and $b \vee \bar{c}$ will increase $\mu(\bar{c})$ by 1, making $\min(\mu(\bar{c}), \mu(c)) = 1$.

Let $F(i)$ be the remaining nontrivial (neither empty nor tautology) clauses when dec_max_2_sat(i, g) in figure 1 is called. Recall that for any unassigned literal x, $\mu(x)$ is the number of the unit clauses for literal x. Let $\mathbf{u}(x)$ be the multiset of $\mu(x)$ copies of the unit clause x and $xB(x)$ denote the set $\{(x \vee y) \mid y \in B(x)\}$. Then we have

$$F(i) = \mathbf{u}(x_i) \cup \mathbf{u}(\bar{x}_i) \cup x_i B(i) \cup \bar{x}_i B\big(\bar{i}\big) \cup F(i+1)$$

where $F(i + 1)$ is the subset of $F(i)$ not containing the variable x_i (assuming $F(n + 1) = \emptyset$).

Let F be $F(i)$ and x be x_i in

$$K(F) = \min\big(K(F_x), K(F_{\bar{x}})\big),$$

we then have

$$K\big(F(i)_{x_i}\big) = \mu\big(\bar{i}\big) + K\big(B\big(\bar{i}\big) \cup F(i+1)\big), \tag{1}$$

$$K\big(F(i)_{\bar{x}_i}\big) = \mu(i) + K\big(B(i) \cup F(i+1)\big), \tag{2}$$

$$K\big(F(i)\big) = \min\big(K\big(F(i)_{x_i}\big), K\big(F(i)_{\bar{x}_i}\big)\big). \tag{3}$$

Since $K(F(i+1)) \leqslant K(Y \cup F(i+1))$ for any clause set Y, we have $\min(\mu(\bar{i}), \mu(i)) + K(F(i + 1)) \leqslant K(F(i))$, by discarding $B(i)$ and $B(\bar{i})$ from (1) and (2), respectively.

This relation shows why LB2 is an admissible lower bound. To show that LB4 is admissible, we need the following lemma.

Lemma 3.6. For any subset $X \subseteq B(i)$, if $|X| \leqslant \mu(\bar{i}) - \mu(i)$, then

$$\min\big(\mu(\bar{i}), \mu(i)\big) + K\big(X \cup F(i+1)\big) \leqslant K\big(F(i)\big). \tag{4}$$

Proof. If $|X| = 0$, then (4) is trivially true. If $|X| > 0$, then $\mu(\bar{i}) > \mu(i)$ and (4) becomes

$$\mu(i) + K\big(X \cup F(i+1)\big) \leqslant K\big(F(i)\big). \tag{5}$$

According to (3), there are two cases:

Case 1: $K(F(i)_{x_i}) < K(F(i)_{\bar{x}_i})$. So $K(F(i)) = K(F(i)_{x_i}) = \mu(\bar{i}) + K(B(\bar{i}) \cup F(i+1))$ by (1). Because $\mu(\bar{i}) \geqslant \mu(i) + |X|$ and $K(B(\bar{i}) \cup F(i+1)) \geqslant K(F(i+1)) \geqslant K(X \cup F(i+1)) - |X|$, we have $K(F(i)) \geqslant \mu(i) + |X| + K(X \cup F(i+1)) - |X|$, or (5).

Case 2: $K(F(i)_{x_i}) \geqslant K(F(i)_{\bar{x}_i})$. So $K(F(i)) = K(F(i)_{\bar{x}_i}) = \mu(i) + K(B(i) \cup F(i+1))$ by (2). Since $X \subseteq B(i)$, $K(B(i) \cup F(i+1)) \geqslant K(X \cup F(i+1))$. So (5) holds. □

A mirror of the above lemma can be obtained by switching i and $\bar{i}$.

Theorem 3.7. LB4 $\leqslant K(F)$.

Proof. We will use lemma 3.6 (or its mirror) at each iteration of i in the procedure lower_bound4. If $i = n$, then LB4 = LB3 = LB2 = $\min(\mu(i), \mu(\bar{i}))$. If $i < n$, suppose LB4′ is the value computed inside lower_bound4 for $X \cup F(i+1)$, where

$$F(i) = \mathbf{u}(x_i) \cup \mathbf{u}(\bar{x}_i) \cup x_i B(i) \cup \bar{x}_i B\big(\bar{i}\big) \cup F(i+1).$$

As an inductive hypothesis, we assume

$$LB4' \leqslant K\big(X \cup F(i+1)\big).$$

If $\mu(\bar{i}) > \mu(i)$ then $Y = B(i)$ and a subset X of Y is added at lines 6–9, where $|X| \leqslant t$ as computed at line 4. Since LB4 = $\min(\mu(\bar{i}), \mu(i)) +$ LB4′, by induction hypothesis and lemma 3.6,

$$\text{LB4} \leqslant \min\big(\mu(\bar{i}), \mu(i)\big) + K\big(X \cup F(i+1)\big) \leqslant K\big(F(i)\big).$$

The case when $\mu(\bar{i}) \leqslant \mu(i)$ is similar. In both cases, LB4 $\leqslant K(F(i))$. □

3.3. Comparison of LB3 and LB4

Since LB3 and LB4 have the same worst-case time complexity, we wish that LB4 is never smaller than LB3, as example 3.5 already showed that LB4 can be greater than LB3.

Let us look at example 3.5 again. We saw that LB4 $= 1$ when $a < b < c$. The same result can be obtained when $b < a < c$, $a < c < b$, or $b < c < a$. However, if $c < a < b$ (or $c < b < a$), then LB4 $= 0$ because neither $\mu(a)$ nor $\mu(b)$ can be increased by the two binary clauses.

This simple example shows that the value of LB4 depends on how the propositional variables are ordered in the implementation. In contrast, the value of LB3 does not depend on variable ordering. Variable ordering is an important performance issue of the algorithm in figure 1. In the next section, we will compare two orderings, including an ordering based on the implication graph of F, and see their impact to the performance.

The next example shows that the ordering for choosing clauses in the computation of LB3 and LB4 also affect their values.

Example 3.8. Let F be represented by μ and S, where $\mu(a) = \mu(b) = \mu(c) = \mu(d) = 0$, $\mu(\bar{a}) = \mu(\bar{b}) = \mu(\bar{c}) = \mu(\bar{d}) = 1$, and $S = \{a \vee b, b \vee c, c \vee d\}$. Obviously, $K(F) = 2$. If $b \vee c$ is chosen first at line 2 of lower_bound3, then LB3 $= 1$; otherwise, LB3 $= 2$. Similarly, for computing LB4, if $b \vee c$ is chosen first at line 6 of lower_bound4 (either b or c must be the smallest in the variable ordering), then LB4 $= 1$; otherwise LB4 $= 2$.

This example shows that LB3 may be 2 while LB4 may be 1. Thus LB3 and LB4 are different in general. Now we have the following question: If the same clause ordering is used for computing both LB3 and LB4, is it true that LB3 $\leqslant$ LB4? The answer is unfortunately no, as illustrated by the following example.

Example 3.9. Suppose F is represented by μ and S, where $\mu(a) = \mu(b) = \mu(c) = \mu(\bar{c}) = 0$, $\mu(\bar{a}) = \mu(\bar{b}) = 1$ and $S = \{a \vee c, a \vee b\}$. Then $K(F) = 1$ and LB3 $= 1$ because of $a \vee b$. However, if the clauses are chosen in the given order, then LB4 $= 0$ because $t = 1$ for a is used on c when $a \vee c$ is chosen.

To avoid the problem illustrated in the above example, at line 6 of lower_bound4, we may prefer those clauses in Y that will increase the value of LB4. This idea is implemented as function lower_bound4a in figure 3. Those literals in Y that will not increase the value of LB4 immediately are stored in $S \subseteq Y$ and processed later at lines 12–14.

However, we do not have a proof that LB4 computed by lower_bound4a is always not less than LB3. To make sure LB4 $\geqslant$ LB3, let us say a clause ordering $<_c$ is LB3-*compatible* if for any two clauses $(a \vee b)$ and $(c \vee d)$, if $(a \vee b)$ satisfies $(c(a) > 0) \wedge (c(b) > 0)$ but $(c \vee d)$ does not, then $(a \vee b) <_c (c \vee d)$.

Theorem 3.10. If we select clauses from Y at line 6 of lower_bound4(i) using a LB3-compatible clause ordering (from small to big), then LB3 $\leqslant$ LB4.

```
function lower_bound4a ( i: integer ) return integer
1  LB4 = LB1;
2  while (i ⩽ n) do
3     LB4 = LB4 + min(μ(ī), μ(i));
4     t = |μ(ī) − μ(i)|;
5     if μ(ī) > μ(i) then Y = B(i) else Y = B(ī)
6     S = ∅
7     for j ∈ Y if (t > 0) do
8        if (μ(j) ⩾ μ(j̄)) then S = S ∪ {j} else
9           t = t − 1; μ(j) = μ(j) + 1;
10       end if
11    end for
12    for j ∈ S if (t > 0) do
13       t = t − 1; μ(j) = μ(j) + 1;
14    end for
15    i = i + 1
16 end while
17 return LB4
end function
```

Figure 3. The procedure for computing LB4a.

Proof. Under the given condition, whenever line 4 of lower_bound3 is executed (LB3 is increased by one), lines 7, 8 of lower_bound4 will be executed, too, to increase LB4 by one. □

In example 3.9, any LB3-compatible ordering will place $a \vee b$ before $a \vee c$. Using this ordering for computing LB4, we have LB3 = LB4 = 1.

The only extra cost for using a LB3-compatible clause ordering in lower_bound4 is that we have to execute lines 6–9 twice. The first time for those $y \in Y$, $c(y) > 0$, and the second time for those such that $c(y) \leqslant 0$. In other words, using this ordering will not compromise the worst-case time complexity of lower_bound4. However, our implementation of LB4 using a LB3-compatible clause ordering is slightly slower than lower_bound4a because of this second visit to Y.

4. Using SCC

Preprocessing is a procedure used before running the MAX-2-SAT algorithm. Our preprocessing is based on strongly connected component (SCC) of implication graph. Given a 2CNF formula F, the implication graph, G_F, of F is a directed graph where the vertices are the set of the literals whose variables appear in F and there is an edge from x to y iff either $\bar{x} \vee y \in F$ or $x = \bar{y}$ and y is a unit clause in F. It is proved in [3] that

F is unsatisfiable iff G_F has a SCC which contains both x and $\bar{x}$ for some literal x. For MAX-2-SAT, we may use SCC to simplify the original problem:

- If a SCC does not contain any conflicting literals, delete the literals in this SCC from the original 2CNF formula.
- If there are more than one SCC, divide the original 2CNF formula according to the SCCs and run the MAX-2-SAT algorithm against each component separately.

The idea of "divide-and-conquer" is not new in SAT. For instance, Bayardo and Pehoushek [6] have used this idea for counting models of SAT. However, it is new to use this idea based on SCC for MAX-2-SAT. While this idea is appealing, in the study of random MAX-2-SAT, we tested thousands of instances but found that very few of them contain more than one SCC with conflict (i.e., with conflicting literals). This should not be a surprise because in the study of random graphs, Erdös and Rnyi [10] showed that for a simple graph with n nodes, when the number m of edges grows from 0 to $n(n-1)/2$, the first cycle appears when $\varepsilon n < m < (1/2 - \varepsilon)n$ and then the graph consists of trees and a few unicyclic components until $m \approx n/2$. When $m = (1+\varepsilon)n/2$, there is a unique giant component and all other components are trees or unicyclic; after that, all other components will merge into the giant component. The application of this result to MAX-2-SAT implication graphs is that only the giant component can contain conflicting literals. That is, it is not likely there is more than one SCC with conflict.

4.1. Preprocessing of clauses

Despite of the fact that we have only one SCC with conflict in most cases, we can combine the above idea with some known pre-processing methods [4,5,13–16,18] to simplify a 2CNF formula before running the decision algorithm. We found that the following sequence of operations is very effective for most 2CNF formulas:

- Unit rule. If $F = \{x \vee y\} \wedge \{x \vee \overline{y}\} \wedge F'$, first let $F = F'$, $\mu(x) = \mu(x) + 1$ and $d = \min(\mu(x), \mu(\bar{x}))$, we then let $K(F) = K(F') + d$, $\mu(x) = \mu(x) - d$, $\mu(\bar{x}) = \mu(\bar{x}) - d$.
- Create the implication graph G_F from F and locate SCCs in G_F. If a SCC does not contain conflicting literals, we delete this component.
- Resolution Rule: If $F = (x \vee y) \wedge (\bar{x} \vee z) \wedge F'$ and F' does not contain x and $\bar{x}$, $K(F) = K(F' \wedge (y \vee z))$.
- Export each SCC as a new 2CNF formula: For each edge $(x \to y)$ and $\mathit{variable}(x) < \mathit{variable}(y)$ in the component, add $\bar{x} \vee y$ in the formula. We also need to copy the value $\mu(x)$ from step 1 to each component.
- Solve each new 2CNF formula by the MAX-2-SAT algorithm.

As shown later in the paper (see table 3), this preprocessing procedure reduces the size of random instance greatly and improves considerably the performance of our algorithm for random MAX-2-SAT instances.

4.2. A new variable ordering

Another new usage of SCC is to design a variable ordering for the algorithm in figure 1. As pointed in section 2, this algorithm uses a fixed ordering, i.e., from 1 to n, to assign truth value to variables. It is found useful in [24] to sort the variables according their occurrences in the input in nonincreasing order and then assign variables in that order. Using SCC, we design a new weight function for variables and we then sort the variables by this weight in nonincreasing order.

The weight function is computed using the following procedure: At first each variable has a weight equal to 0. Then we update the weight by finding the shortest path between every pair of $(x, \bar{x})$ in the SCC. If the path goes through node y to z, we increase both the weights of y and z by 1.

The intuition behind this ordering is that we want to derive conflicting clauses as early as possible so that the lower bound checking at line 2 of the algorithm in figure 1 can be more effective. If a clause appears in the shortest path connecting two conflicting literals in a SCC, we give the literals in this clause higher weights so that the truth value of this clause can be decided early. The experimental results in the next section show that the ordering by the new weight function performs better than the occurrence ordering when either m/n or $K(F)$ is small.

5. Experimental results

We have implemented all three techniques presented in this paper to support the algorithm presented in figure 1. The implementation is written in C++ and experimental results seem promising. All data are collected on a cluster of Pentium 4 2.4 GHz machines each with 1 GB memory. All these machines run Red Hat Linux 9.0 with gcc version 3.2.2. Note that the running time of the DIMACS hardware program "dfmax r500.5.b" is 13.7 on these machines.

5.1. Comparison of different implementations

Table 1 shows some results of Borchers and Furman's program (BF) [5], Alsinet et al.'s (AMP, the option LB2 − I + JW is used), and our implementations (LB3, LB4, LB4a) on the MAX-2-SAT problems distributed by Borchers and Furman [5].[1] The benchmarks contain randomly generated 2-CNFs of 50, 100, and 150 variables (#var), and are available from their Web site. Note that g_0 in our algorithm is at first set to the number found by the first phase of Borchers and Furman's local search procedure and then decreased by one until the optimal value is decided as BF and AMP did. It is clear from the table that LB4a provides the best result while LB3 and LB4 are ranked second. The harder the problem, the more significant the gain of LB4a over other implementa-

[1] Since we could not obtain Zhao and Zhang's implementation [25] and Gramm's implementation [11], we could not include their run times in our experiment.

Table 1
Experimental results on Borchers and Furman's examples. (in seconds).

Problem		False	BF	AMP	LB3	LB4	LB4a
#var	#cls	clauses					
50	100	4	0.02	0.03	0.02	0.02	0.01
	150	8	0.06	0.03	0.02	0.02	0.02
	200	16	4.18	0.38	0.03	0.03	0.03
	250	22	24	0.26	0.05	0.04	0.04
	300	32	350	4.88	0.09	0.09	0.07
	350	41	2556	10	0.09	0.17	0.11
	400	45	2308	4.65	0.05	0.08	0.06
	450	63	–	44	0.23	0.32	0.21
	500	66	–	17	0.12	0.17	0.15
100	200	5	0.14	0.16	0.03	0.03	0.04
	300	15	501	29	0.42	0.66	0.4
	400	29	–	1204	1.26	3.15	0.98
	500	44	–	–	192	180	62
	600	65	–	–	303	394	130
150	300	4	0.18	0.22	0.12	0.10	0.08
	450	22	–	–	64	19	7.11
	600	38	–	–	347	385	54

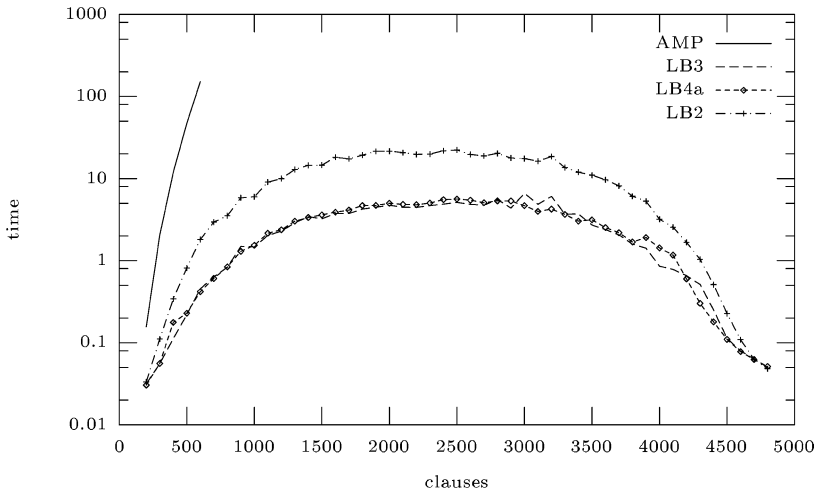

Figure 4. Running time for Alsinet et al.'s program, LB2, LB3 and LB4a for 50 variables problems.

tions. All of our implementations are much faster than Borchers and Furman's program and Alsinet et al.'s modification.

Figures 4 and 5 compare Alsinet et al.'s program and our implementations (LB2, LB3, LB4a) on the random MAX-2-SAT problems of 50 variables and 100 variables, respectively. We considered the following cases: $n = 50$ variables with $m = 200$, $300, 400, \ldots, 4700, 4800$ clauses and $n = 100$ variables with $m = 200, 250, \ldots,$

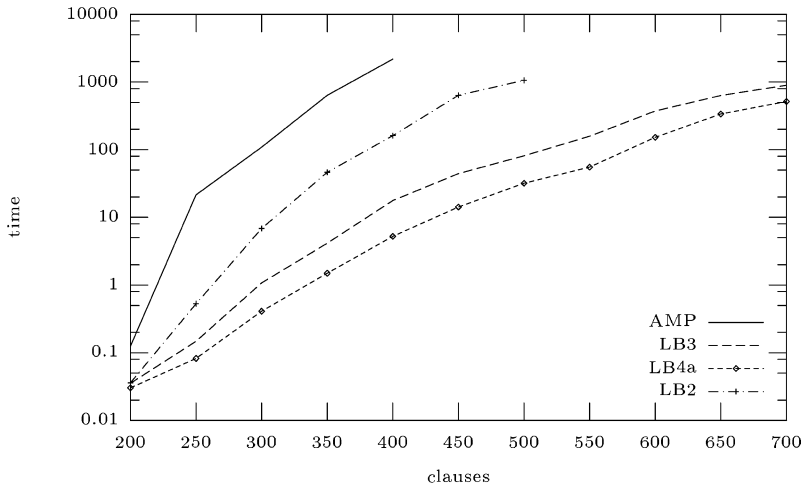

Figure 5. Running time for Alsinet et al.'s program, LB2, LB3 and LB4a for 100 variables problems.

Table 2
Performance comparison for LB3 and LB4a. The run times are in seconds. The number in the parentheses following the run time is the number of unfinished cases after two hours of running.

Problem		LB3		LB4a	
#var	#cls	#branches	sec	#branches	sec
100	200	3.7e+03	0.035	1.63e+03	0.030
	250	6.22e+04	0.15	1.39e+04	0.082
	300	5.91e+05	1.08	1.07e+05	0.41
	350	2.28e+06	4.14	3.84e+05	1.5
	400	9.5e+06	17	1.09e+06	5.23
	450	1.77e+07	44	3.14e+06	14
	500	3.91e+07	81	7.32e+06	32
	550	7.32e+07	158	1.27e+07	55
	600	1.69e+08	373	3.38e+07	152
	650	3.15e+08	632	6.85e+07	337
	700	4.34e+08	1191	1.14e+08	512
200	400	1.61e+06	3.41	2.09e+05	0.88
	420	1.27e+07	28	1.3e+06	5.33
	440	5.79e+07	145	3.17e+06	15
	460	1.24e+08	330 (3)	1.81e+07	90
	480	1.46e+08	405 (10)	2.26e+07	117 (1)
	500	2.69e+08	758 (21)	4.31e+07	244 (2)

650, 700 clauses. For each case, we generated 100 random problems (excluding satisfiable ones). Both figures show that all our programs are much faster than Alsinet et al.'s program, and we can see that LB3 and LB4a are consistently better than LB2. Note that in the figure of $n = 50$ variables, the running time of our program is decreasing while $c = m/n$ is increasing when $c = m/n$ is large enough. The reason for this is that the preprocessing (unit rule) can reduce a lot of clauses when c is very large.

Table 3
Performance comparison for SOLB3 and SWLB3 (reduced is the problem after preprocessing).

Problem		Reduced		SOLB3	SWLB3
#vars	#clauses	#vars	#clauses	branches	branches
300	420	56.6	79.6	1.19e+03	326
300	450	77.4	115	1.66e+04	3.32e+03
300	480	102	166	2.01e+05	1.73e+05
300	510	122	210	5.87e+06	1.99e+06
300	540	139	253	2.01e+07	8.66e+06
500	650	59	76	1.3e+03	1e+03
500	700	89	120	1.7e+05	1.5e+04
500	750	132	197	5.9e+06	1.2e+06
500	800	171	277	3.1e+07	1.9e+07

5.2. *Comparison of LB3 and LB4a*

From figures 4 and 5, we see that for $n = 50$, LB3 and LB4a are almost as fast as each other. For harder problem, $n = 100$, LB4a is faster than LB3.

Table 2 provides a detailed comparison of LB3 and LB4a. The instances are random 2CNF with 100 or 200 variables and various number of clauses. The data show the average run times on 100 instances (of the same numbers of variables and clauses). It is clear from the table that LB4a explores only 1/4 of the search space (represented by the number of branches in the search tree) of LB3 and LB4a is about 3 times faster than LB3 on average.

5.3. *Comparison of variable orderings*

We have run our program with two different configurations: sorting variables by occurrence with LB3 (SOLB3), sorting variables by weight with LB3 (SWLB3). Table 3 and figures 6 and 7 show the results of the two configurations on random MAX-2-SAT problems with 300, 500 variables. Each case has 100 random instances. We can also see that SWLB3 is faster than SOLB3 when $c = m/n$ is relatively small.

5.4. *Max-Cut*

Given an undirected simple graph $G = (V, E)$, where $V = \{x_1, \ldots, x_n\}$ is the set of vertices and E is the set of edges, let weight w_{x_i,x_j} be associated with each edge $(x_i, x_j) \in E$. The Weighted Max-Cut problem is to find a subset S of V such that

$$W\big(S, \overline{S}\big) = \sum_{x_i \in S, x_j \in \overline{S}} w_{x_i,x_j}$$

is maximized, where $\overline{S} = V - S$. In this paper, we let weight $w_{x_i,x_j} = 1$ for all edges. The following theorem shows how to reduce the Max-Cut problem to the MAX-2-SAT problem.

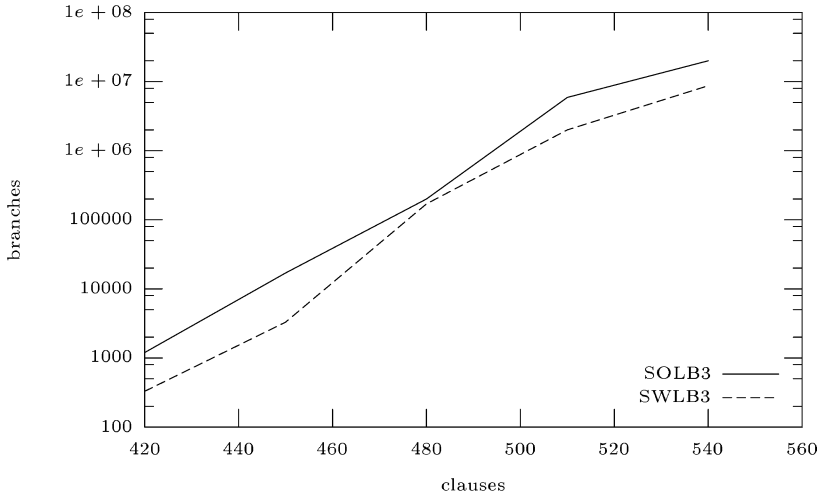

Figure 6. Performance comparison for SOLB3 and SWLB3 for $n = 300$.

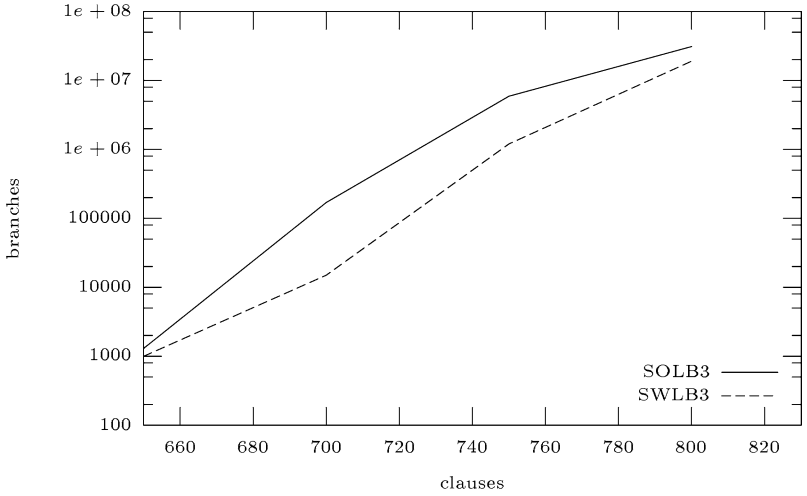

Figure 7. Performance comparison for SOLB3 and SWLB3 for $n = 500$.

Theorem 5.1 [7,17]. Given $G = (V, E)$, where $|V| = n$, $|E| = m$, we assume weight $w(x_i, x_j) = 1$ for each $(x_i, x_j) \in E$. We construct an instance of MAX-2-SAT as follows. Let V be the set of propositional variables and for each edge $(x_i, x_j) \in E$, we create exactly two binary clauses: $(x_i \vee x_j)$ and $(\bar{x}_i \vee \bar{x}_j)$. Let F be the collection of such binary clauses, then the Max-Cut problem has a cut of weight k iff the MAX-2-SAT problem has an assignment under which $m + k$ clauses are true.

We have run thousands instances of the random Max-Cut problem. Some results are shown in figures 8 and 9. In each case, there are 100 random instances. For the instances where the number of variables $n = 50$, SO (sorting variables by occurrence)

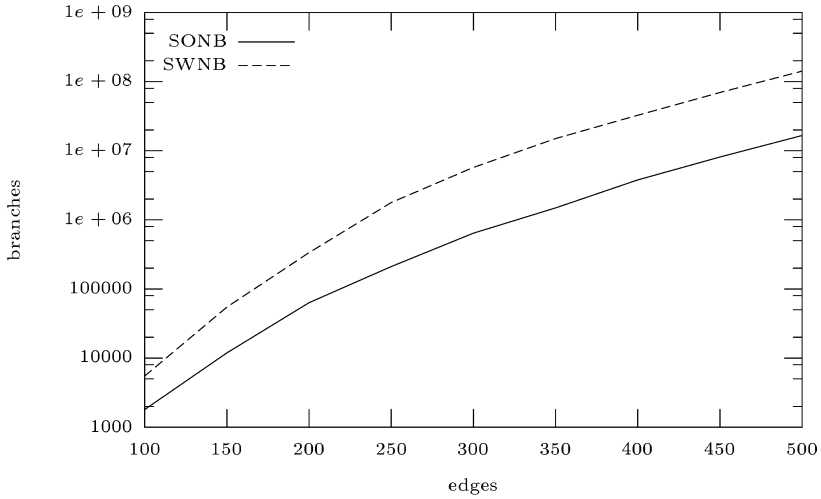

Figure 8. Performance of SOLB3 and SWLB3 for Max-Cut problems ($n = 50$).

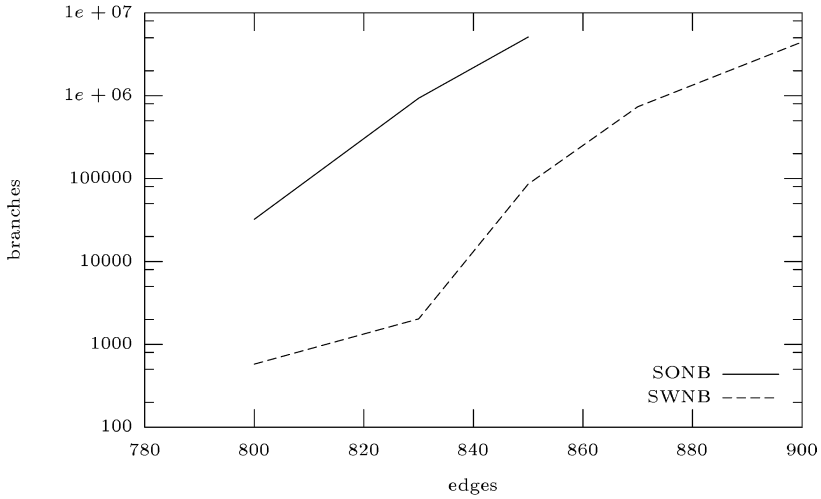

Figure 9. Performance of SOLB3 and SWLB3 for Max-Cut problems ($n = 1000$).

is better than SW (sorting variables by weight). Those instances have a relatively large $c = m/n > 2$ in most cases. For the instances with $n = 1000$, SW is often more than 10 times faster than SO when $c = m/n < 2$.

6. Conclusion

We have studied three new lower bound functions (LB3, LB4 and LB4a) for the branch-and-bound algorithm of MAX-2-SAT and proved that these bounds are indeed admissible and tighter than existing lower bound functions. Experimental results showed

that these new lower bound functions are consistently better than other lower bound functions. We have also used SCCs of the implication graph of a 2CNF formula to simplify the formula and to design a new weight for variables. The experiments showed that sorting variables by weight performs much better when the clause-to-variable ratio, $c = m/n$, is less than 2. The proposed preprocessing technique can reduce the size of the input greatly no matter what is the clause-to-variable ratio. A direct outcome of this research is a high-performance implementation of an exact algorithm for MAX-2-SAT, which outperforms any implementation we know about in the same category. We applied the MAX-2-SAT algorithm to solve large instances of the Max-Cut problem and the experimental results showed that this approach is feasible and effective. All the techniques presented in the paper can be applied to the weighted MAX-2-SAT problem where each clause has a weight. The Max-Cut problem with arbitrary weights can be easily converted into an instance of the weighted MAX-2-SAT. As future work, we will specialize and improve the MAX-2-SAT algorithm for the Max-Cut problem and attack some real world Max-Cut problems. We will extend the techniques presented in the paper to solve general MAX-SAT problems. We are also interested in applying these techniques to approximate methods for MAX-2-SAT.

References

[1] T. Alsinet, F. Manyà and J. Planes, Improved branch and bound algorithms for Max-SAT, in: *Proc. of 6th International Conference on the Theory and Applications of Satisfiability Testing, SAT2003* (2003) pp. 408–415.

[2] T. Alsinet, F. Manyà and J. Planes, Improved branch and bound algorithms for Max-2-SAT and Weighted Max-2-SAT, in: *Catalonian Conference on Artificial Intelligence* (2003).

[3] B. Aspvall, M.F. Plass and R.E. Tarjan, A linear-time algorithm for testing the truth of certain quantified Boolean formulas, Information Processing Letters 8(3) (1979) 121–123.

[4] N. Bansal and V. Raman, Upper bounds for MaxSat: Further improved, in: *Proc. of the 10th Annual Conference on Algorithms and Computation, ISSAC'99*, eds. Aggarwal and Rangan, Lecture Notes in Computer Science, Vol. 1741 (Springer, New York, 1999) pp. 247–258.

[5] B. Borchers and J. Furman, A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems, Journal of Combinatorial Optimization 2(4) (1999) 299–306.

[6] R.J. Bayardo and J.D. Pehoushek, Counting models using connected components, in: *17th National Conference on Artificial Intelligence (AAAI)* (2000) pp. 157–162.

[7] J. Cheriyan, W.H. Cunningnham, L. Tuncel and Y. Wang, A linear programming and rounding approach to Max 2-Sat, in: *Discrete Mathematics and Theoretical Computer Science,* Vol. 26 (Science Press, New York, 1996) pp. 395–414.

[8] E. Dantsin, A. Goerdt, E.A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan and U. Schöning, A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search, Theoretical Computer Science 289(1) (2002) 69–83.

[9] M. Davis, G. Logemann and D. Loveland, A machine program for theorem-proving, Communications of the Association for Computing Machinery 7 (1962) 394–397.

[10] P. Erdös and A. Rnyi, On the evolution of random graphs, Mat. Kutato Int. Kozl. 5 (1960) 17–61.

[11] J. Gramm, Exact algorithms for Max2Sat and their applications, Diplomarbeit, Universität Tübingen (1999).

[12] S. Givry, J. Larrosa, P. Meseguer and T. Schiex, Solving Max-SAT as weighted CSP, in: *Principles and Practice of Constraint Programming – 9th International Conference CP 2003*, Kinsale, Ireland (September 2003).
[13] J. Gramm, E.A. Hirsch, R. Niedermeier and P. Rossmanith, Worst-case upper bounds for MAX-2-SAT with application to MAX-CUT, Discrete Applied Mathematics 130(2) (2003) 139–155.
[14] P. Hansen and B. Jaumard, Algorithms for the maximum satisfiability problem, Computing 44 (1990) 279–303.
[15] E.A. Hirsch, A new algorithm for MAX-2-SAT, in: *Proc. of the 17th International Symposium on Theoretical Aspects of Computer Science, STACS 2000*, Lecture Notes in Computer Science, Vol. 1770 (Springer, New York, 2000) pp. 65–73.
[16] E.A. Hirsch, New worst-case upper bounds for SAT, Journal of Automated Reasoning 24(4) (2000) 397–420.
[17] M. Mahajan and V. Raman, Parameterizing above guaranteed values: Max-Sat and Max-Cut, Journal of Algorithms 31 (1999) 335–354.
[18] R. Niedermeier and P. Rossmanith, New upper bounds for maximum satisfiability, Journal of Algorithms 36 (2000) 63–88.
[19] H. Shen and H. Zhang, An empirical study of Max-2-Sat phase transitions, in: *Proc. of LICS'03 Workshop on Typical Case Complexity and Phase Transitions*, Ottawa, Canada (June 2003).
[20] H. Shen and H. Zhang, Improving Exact Algorithms for MAX-2-SAT, in: *Proc. of 8th International Symposium on Artificial Intelligence and Mathematics* (2004).
[21] H. Shen and H. Zhang, Study of Lower Bound Functions for MAX-2-SAT, in: *Proc. of the 19th National Conf. on Artificial Intelligence (AAAI)*, San Jose, CA (July 2004).
[22] R. Wallace and E. Freuder, Comparative studies of constraint satisfaction and Davis–Putnam algorithms for maximum satisfiability problems, in: *Cliques, Coloring and Satisfiability*, eds. D. Johnson and M. Trick (1996) pp. 587–615.
[23] H. Xu, R.A. Rutenbar and K. Sakallah, Sub-SAT: A formulation for related Boolean satisfiability with applications in routing, in: *ISPD'02*, San Diego, CA (April 2002).
[24] H. Zhang, H. Shen and F. Manyà, Exact algorithms for MAX-SAT, in: *Proc. of International Workshop on First-Order Theorem Proving (FTP 2003)*.
[25] X. Zhao and W. Zhang, An efficient algorithm for maximum Boolean satisfiability based on unit propagation, linear programming, and dynamic weighting, Preprint, Department of Computer Science, Washington University (2004).